Home **Security** Sweet Home

Projet innovant

Avril 2011

Home Security Sweet Home

La bible

Xavier Dupessey
Hugo Lafaye de Micheaux
Enseignant responsable: Didier Donsez

Sommaire

Présentation

Le projet

L'équipe

Modules utilisés

iButton

Dispositifs distants

Arduino receiver

E-mail sender

Webcam / video

Calendar

Surveillance Brain

Simulation Diasuite

Diasuite

1. Définition du design

2. Implémentation des classes de appareils, contextes et contrôleurs

3. Édition de la simulation ou phase de test

4. Lancement de la simulation

Présentation

Le projet

HSSH est un projet réalisé dans le cadre de notre 4ème année à Polytech Grenoble, filière RICM (Réseaux Informatiques et Communication Multimédia). Le but de ce projet est de nous permettre la mise en pratique des connaissances acquises au cours de notre formation.

Il nous permet de mettre en application nos cours de génie logiciel, gestion de projet, programmation orientée objet, etc.

HSSH : Home Security Sweet Home

La salle AIR (Ambiance Intelligent Room) est une pièce contenant de nombreux objets intelligents et innovants. Elle est située au 2ème étage de l'école Polytech Grenoble (n°259) et est réservée aux étudiants pour la réalisation de projets dans le cadre de leurs études. Étant donné la haute technologie présente dans cette salle, il est impératif d'y installer un système de surveillance.

Le but de ce projet est d'assurer la surveillance de cette salle par le biais d'un réseau de capteurs ainsi que des informations factuelles tel que le planning de la salle. Différents dispositifs seront déclenchés selon le problème (ou l'intrusion) détecté.

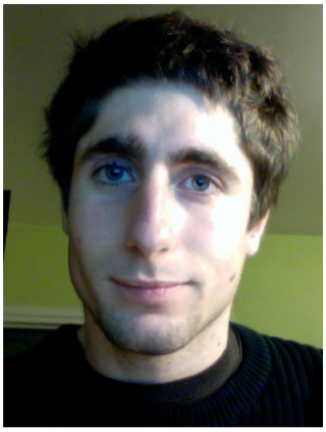
Un projet innovant

HSSH est un projet innovant : aujourd'hui, il existe encore très peu de systèmes permettant de centraliser toute la sécurité d'une pièce ou d'une maison, et cela à distance.

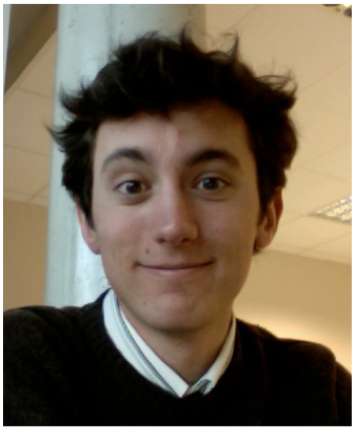
L'intérêt pour nous est de concevoir et créer un système relativement flexible et évolutif. Par exemple, il faut pouvoir intégrer de nouveaux types de capteurs aisément.

L'équipe 2011

Notre équipe est constituée de 2 personnes. Nous sommes tous les 2 en option Communication Multimédia, dans la filière RICM (4ème année) à Polytech Grenoble.



Xavier Dupessey
(chef de projet)



Hugo Lafaye De Micheaux

Enseignant responsable : Didier Donsez

Modules utilisés

iButton

Présentation

Un iButton, malgré son nom, s'apparente plus à une clé électronique. Il intègre une puce électronique qui permet de stocker des données (1Ko de EEPROM pour le DS1972), accessible via le protocole 1-wire.

Afin de le connecter à l'ordinateur, nous utilisons l'adaptateur ibutton - usb DS9490R.

un iButton

Adaptateur ibutton usb - DS9490R

Côté logiciel : OWFS

La société qui créer ces ibuttons ne propose pas de drivers officiels pour Linux. Il existe cependant une alternative, le projet OWFS.

OWFS permet aux périphériques utilisant le protocole 1-wire d'apparaitre comme des fichiers dans un répertoire. Par exemple, si on utilise un capteur de température 1-wire, la commande "cat */temperature" permet de visualiser toutes les mesures. Chaque périphérique dispose ainsi de son propre répertoire qui regroupe ses propriétés sous forme de différents fichiers.

Installation

Depuis peu de temps, des packages .deb sont disponibles pour Ubuntu et Debian. Vous les trouverez sur le dépot OWFS de Davromaniak.

1. Ajouter le dépot correspondant à votre distribution Linux au fichier */etc/apt/sources.list*
2. Mettre à jour la liste des dépots : *sudo apt-get update*
3. Installer OWFS : *sudo apt-get install owfs*
4. Créer le répertoire */mnt/1wire* : *sudo mkdir /mnt/1wire*

Utilisation

Lancement de OWFS avec la commande :

```
sudo owfs --allow_other -u --timeout_directory=1 /mnt/1wire
```

Le paramètre *timeout_directory* permet de spécifier l'intervalle de temps en seconde de la découverte des périphériques 1-wire connectés. On la met à *1* pour que le ibutton soit reconnu dès sa connexion à l'ordinateur.

Exemple d'utilisation :

```
xavier@xavier-linux ~ $ ls /mnt/1wire/
81.58572C000000  bus.0  settings  statistics  structure  system  uncached
```

Le répertoire *81.58572C000000* correspond à l'adaptateur DS9490R en lui-même.

Maintenant, connectons un ibutton :

```
xavier@xavier-linux ~ $ ls  /mnt/1wire/
2D.C29DCC000000  81.58572C000000  bus.0  settings  statistics  structure  system  uncached
```

Nous constatons que le répertoire *2D.C29DCC000000* a été créé : il correspond à notre ibutton.

Lien avec Java

Pour détecter la connexion / déconnection d'ibuttons avec Java, nous surveillons simplement la création / suppresion des répertoires effectué par OWFS. Voir paquages *hssh.ibutton* et *hssh.util.directorywatcher*.

Intégration dans le projet

Dans ce projet, ils sont utilisés pour identifier une personne. Dans chaque iButton, nous sauvegardons un identifiant unique. Le système de surveillance connaît la liste de ces identifiants ainsi que les noms des personnes associées (via un fichier xml).

```
<iAccounts>
    <iAccount type="administrator" ibuttonId="2D.C29DCC000000" name="Xavier"/>
    <iAccount type="administrator" ibuttonId="2D.9EBACC000000" name="Hugo"/>
    <iAccount type="guest" ibuttonId="2D.5D3ECD000000" name="Quelqu un"/>
</iAccounts>
```

Ainsi, lorsqu'une intrusion est détectée, le système de surveillance demande à l'utilisateur de s'identifier. Ce dernier connecte alors son iButton. Si l'utilisateur est reconnu et autorisé par le système, l'alarme ne se déclenche pas.

Dispositifs distants

Les dispositifs distants sont tous les capteurs présents dans la salle qui envoient des messages (leur état, leur niveau de batterie) au système de surveillance (via le module Arduino Receiver). Pour cela, ils disposent d'un module RF émetteur 433MHz.

Il peut donc s'agir de contacteurs, de détecteurs de fumée, ou encore de détecteurs de présence :

Détecteur de fumée

Contacteur

Protocole de communication

Afin de transmettre leurs informations au système de surveillance, nous avons défini un protocole de communication basique. Il est unidirectionnel : les capteurs peuvent uniquement envoyer des informations, mais en aucun cas en recevoir.

Chaque information envoyée est composée d'un seul octet. Celui-ci renseigne l'ID du capteur expéditeur ainsi que son message.

XXXX XXXX

ID du capteur expéditeur (unique)

contenu du message (selon dispositif)

Cela permet d'utiliser 32 capteurs différents dans la salle (2^5 bits).
Utiliser 3 bits est suffisant pour les messages à transmettre car il s'agit, pour la plupart, d'informations de type tout ou rien.

Trames des dispositifs distants

Trames communes

```
xxxx xXxx
      0 = battery state
      1 = other message

xxxx x0XX
       00 = critically empty
       01 = soon empty
       10 = ok
       11 = full
```

Contacteur

```
xxxx x1xX
        0 = close
        1 = open
```

Détecteur de fumée

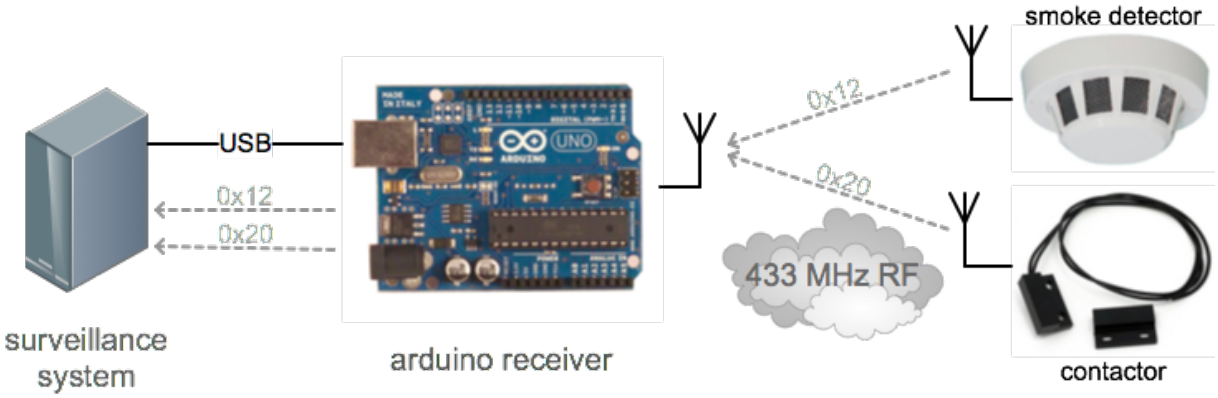
```
xxxx x1xX
        0 = no smoke
        1 = smoke detected
```

Détecteur de présence

```
xxxx x1xX
        0 = no presence
        1 = presence detected
```

Arduino receiver

Ce module est chargé de recevoir tous les messages bruts envoyés par les dispositifs distants (détecteur de fumée, contacteurs, etc). Pour cela, il est composé d'un module RF récepteur 433MHz et est connecté au système de surveillance en USB.



Communication : dispositifs distants - arduino receiver

Communication USB en Java : RXTX

Sous Linux, la communication série est possible en Java avec l'utilisation de la librairie RXTX.

Installation

Télécharger ici : http://rxtx.qbang.org/wiki/index.php/Download

Déplacer RXTXcomm.jar dans /usr/lib/jvm/java-XXX/jre/lib/ext/

Déplacer librxtxSerial.so dans /usr/lib/jvm/java-XXX/jre/lib/i386/

Utilisation

Tout d'abord, il faut savoir où est monté le Arduino receiver. Pour cela, connectez-le à l'ordinateur et exécutez la commande *sudo dmesg | grep tty* :

```
xavier@xavier-linux ~ $ sudo dmesg | grep tty
[    0.000000] console [tty0] enabled
[   24.597257] cdc_acm 3-1:1.0: ttyACM0: USB ACM device
```

Ici, il est monté sur ttyACM0, il faut donc renseigner */dev/ttyACM0* au constructeur de *RemoteDeviceReader*.

⚠ La vitesse de communication est limitée par la carte Arduino Uno à 9600 bauds/sec et est également à définir dans le programme Arduino.

Intégration dans le projet

Dans notre projet, le module Arduino receiver est géré par la classe *RemoteDeviceReader*. Son but est de recevoir tous les messages bruts des dispositifs distant, de les convertir en *RemoteDeviceMessage* et de les transmettre au système de surveillance.

Le système connaît la liste des dispositifs distants grâce à un fichier xml, rensigné manuellement :

```
<?xml version="1.0" encoding="UTF-8" ?>
<remotedevices>
```

```
    <remotedevice type="Contactor" id="1" name="Main door" />
    <remotedevice type="Contactor" id="2" name="North window" />
    <remotedevice type="Contactor" id="3" name="South window" />
</remotedevices>
```

Dans ce fichier, chaque ID doit être unique et compris entre 0 et 31, comme nous l'avons vu sur la page dispositifs distants.

Ajout de nouveaux types de capteurs

Il est très facile d'ajouter de nouveaux types de capteurs : dans le fichier xml, l'attribut *type* de chaque *remotedevice* est également le nom de la classe à appeler pour la création de chaque dispositif. Ainsi, pour ajouter un détecteur de présence (par exemple), il suffit d'ajouter la ligne suivante

```
<remotedevice type="PresenceDetector" id="4" name="Presence detector (living room)" />
```

Et créer les classes *PresenceDetector* et *PresenceDetectorMessage* qui héritent de *RemoteDevice* et *RemoteDeviceMessage*. Consulter la javadoc pour plus d'infos. Voir packages *hssh.devices* et *hssh.devicesmessages*.

Réaliser la communication sans-fil [À FAIRE]

La liaison sans fil n'a pas été réalisée pour cause d'un manque de temps et de connaissances en électronique. Actuellement, le module arduino receiver se contente d'envoyer des mocks.

Liens utiles

RXTX

http://www.developpez.net/forums/d871052/java/general-java/apis/rxtx-port-serie-portinuseexception/
https://bugs.launchpad.net/ubuntu/+source/rxtx/+bug/367833

RF communication

http://www.seeedstudio.com/depot/433mhz-rf-link-kit-p-127.html
http://www.design.ucla.edu/senselab/node/389
http://www.glacialwanderer.com/hobbyrobotics/?p=291

E-mail sender

Le système de sécurité doit être dans la mesure d'envoyer des e-mails. Cela peut être utile, par exemple, pour avertir le propriétaire de la salle lors d'une intrusion.

Nous avons donc créé un "e-mail sender" en Java.

Création d'un compte gmail dédié

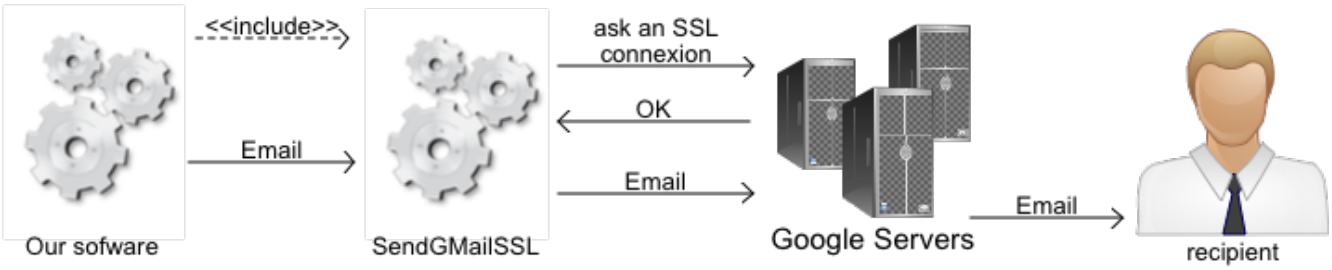
Envoyer un e-mail depuis un programme nécessite de disposer d'un serveur mail. Cela implique de disposer d'un nom de domaine, d'un serveur de nom, etc. Pour éviter l'installation d'un tel système, nous avons décidé d'envoyer nos e-mails depuis les serveur de Google.

Nous avons donc créé un compte gmail : **hssh.notifier *(at)* gmail *(point)* com**

Avec le mot de passe qui va bien.

L'e-mail sender Java

L'envoi des e-mail est assuré par la classe hssh.util.email.SendGMailSSL. Après s'être identifié auprès de cette classe, elle ouvre une connexion SSL sécurisée avec un des serveurs de Google.



Email sender communication

La méthode sendMail, qui prend un objet de type Email en argument, permet d'effectué l'envoi.

Utilisation dans notre projet

Exemple d'un envoi d'email:

```
SendGMailSSL mailserv = new SendGMailSSL("hssh.notifier", <password>);
mailserv.sendMail(new Email("Subject", "Message body", "
 recipient@domain.com"));
```

Webcam / video

De nombreuses bibliothèques permettent de gérer la vidéo / les webcams en Java. Malheureusement, nombre d'entre elles sont des usines à gaz tant au niveau de l'installation, de la configuration ou de l'utilisation.

Nous nous sommes intéressés à :

1. JMF (API permettant la manipulation de vidéos et fichiers audio fournie par Sun), plus mis à jour depuis 2003
2. FMJ (une alternative à JMF)
3. QTJ (QuickTime for Java), non compatible avec Linux
4. VLCJ (java Bendings for VideoLAN), non compatible avec Linux
5. ...

Finalement, nous avons trouvé notre bonheur avec les librairies V4L4J (Video for Linux for Java) et Xuggle.

V4L4J pour la webcam

Installation

Exécution des commandes :

```
sudo apt-get install libv4l4j-java
sudo apt-get udpate
sudo add-apt-repository ppa:gillesg/ppa
```

La version installée chez nous est : *0.8.9-0ubuntu1~ppa4*

Configuration

V4L4J utilise la librairie système *libv4l4j.so*, qui a normalement été placée dans le répertoire */usr/lib/jni/*

Il est nécessaire que le programme java sache où cette librairie se trouve. Lui indiquer avec l'argument suivant (concerne la JRE) :

```
-Djava.library.path=/usr/lib/jni
```

Utilisation

Nous avons été agréablement surpris par la qualité de la documentation et des exemples de V4L4J. Pour l'utiliser plus aisément (avec une Webcam), nous avons créé la classe *hssh.util.webcam.Webcam*.

```
int width = 640, height = 480, quality = 85;
Webcam webcam = new Webcam("/dev/video0");
webcam.init(width, height, quality);
webcam.start();

[...]
BufferedImage img = webcam.getImage();
[...]

webcam.stop();
```

Xuggle pour les enregistrements vidéos

Xuggle [website - wiki] est une bibliothèque Java qui permet de décoder, modifier, encoder et enregistrer des médias.

Installation

Suivre les étapes expliquées ici même : http://www.xuggle.com/xuggler/downloads/

Configuration

De même que pour V4L4J qui nécessite *libv4l4j.so*, xuggle nécessite des librairies présentes dans */usr/local/xuggler/lib*.

L'argument à fournir à la JRE devient donc :

```
-Djava.library.path=/usr/lib/jni:/usr/local/xuggler/lib
```

Utilisation

Dans notre cas, nous utilisons Xuggler pour créer une vidéo à partir d'objets de type BufferedImage (issus de la webcam). Pour faire cela de manière facile, nous avons créé 2 classe : *Video* et *WebcamVideo* (voir package *hssh.util.webcam*).

Voici un exemple d'utilisation :

```
int width = 640, height = 480, quality = 85, framerate = 5, videoduration = 10;
Webcam webcam = new Webcam("/dev/video0");
webcam.init(width, height, quality);
webcam.start();
WebcamVideo wv = new WebcamVideo(webcam, "/home/xavier/Desktop/video.mp4", videoduration, framerate);
wv.startRecord();
webcam.stop();
```

La vidéo enregistrée est au format mp4.

Calendar

Nous avons prévu que les évènements captés n'aient pas les mêmes répercutions suivant l'heure qu'il est. Par exemple, une fenêtre ouverte à 15h un mardi n'est pas important. Mais si on est dimanche à 23h, cela est louche et il faut mieux prévenir le propriétaire de la salle.

Étude de l'existant

Nous avons tout d'abord pensé à utiliser un système de calendrier existant, tel que Google Agenda, ADE, ou encore iCal. Nous avons rapidement abandonné cette idée car cela impliquait de renseigner les horaires de la salle en respectant des règles très précises pour qu'elles puissent être comprises par notre système. La plupart des utilisateurs n'auraient pas compris ou n'auraient pas pris de temps de renseigner de telles informations.

Mise en place d'un Week Calendar

Nous avons créer un objet *WeekCalendar* qui permet de renseigner les horaires d'ouverture de la salle suivant les jours de la semaine. Pour cela, il suffit de renseigner un fichier XML.

Exemple pour le calendrier d'ouverture :

- lundi : 8h05 - 18h
- mardi : 8h30 - 19h
- mercredi : (fermé toute la journée)
- jeudi : 7h50 - 12h
- vendredi : 8h50 - 20h30
- samedi : (fermé toute la journée)
- dimanche : (fermé toute la journée)

Le fichier XML sera :

```
<?xml version="1.0" encoding="UTF-8" ?>
<weekcalendar>
    <day type="opened" name="monday">
        <hours type="begin" hour="8" min="05" />
        <hours type="end" hour="18" min="0" />
    </day>
    <day type="opened" name="tuesday">
        <hours type="begin" hour="8" min="30" />
        <hours type="end" hour="19" min="0" />
    </day>
    <day type="closed" name="wednesday" />
    <day type="opened" name="thursday">
        <hours type="begin" hour="7" min="50" />
        <hours type="end" hour="12" min="0" />
    </day>
    <day type="opened" name="friday">
        <hours type="begin" hour="8" min="50" />
        <hours type="end" hour="20" min="30" />
    </day>
    <day type="closed" name="saturday" />
    <day type="closed" name="sunday" />
</weekcalendar>
```

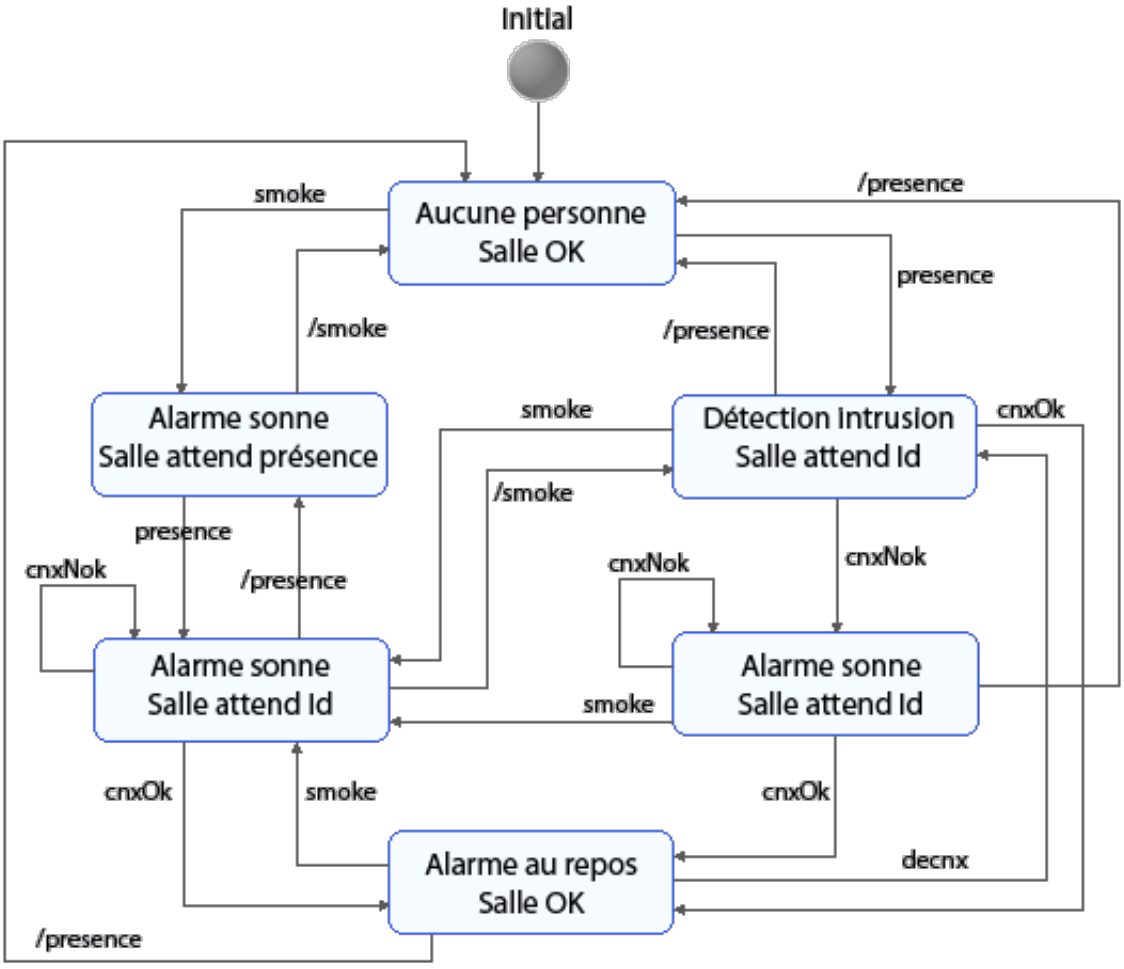
Surveillance Brain

L'automate

Le *Surveillance Brain* est le module central de notre application. C'est lui qui reçoit les évènements transmis par les capteurs et qui déclenche les actions à entreprendre.

Pour réaliser ce système, nous nous sommes basés sur un automate. Les avantages sont multiples :

- simplification de la gestion des états
- simplicité d'évolution
- rapidité de déploiement
- adapté au langage JAVA



Réalisation Java

La classe *Brain* contient l'automate et dispose de :

- la liste des états de l'automate
- l'état courant
- une liste d'actions à effectuer suivant l'état courant

La classe *State* dispose de :

- un identifiant unique (un nombre)
- le fait qu'il soit état initial ou pas (booléen)
- une liste de transition

La classe Transition dispose de :

- un évènement (transmis par les capteurs)
- un état suivant

Pour plus d'informations, consulter la documentation du package *hssh.surveillance*

Simulation Diasuite

Diasuite

Introduction

DiaSuite est un environnement de développement dédié aux applications qui souhaite faire interagir des entités avec un environnement. On appelle cela de nos jours l'« Intelligence Ambiante ».

Cet ensemble d'outils a été et est développé par l'INRIA et le LaBRI (Laboratoire Bordelais de Recherche en Informatique) du Phoenix Research Group.

Il s'applique dans de nombreux domaines tels que : les télécommunications, les systèmes de surveillance, les systèmes d'assistance aux personnes handicapées, ...

L'utilisation de DiaSuite

L'utilisation de DiaSuite se fait en 4 étapes qui sont les suivantes (il faut bien sûr au préalable le télécharger sur le site et l'installer) :

1. définir le design, c'est-à-dire l'architecture du système, par un langage spécifique
2. implémenter les différentes classes et méthodes générées automatiquement après la définition du design
3. tester par un simulateur en 2D les spécifications
4. utiliser le système

Application au système de surveillance HSSH

Dans le système de surveillance Home Security Sweet Home, il y a de nombreux capteurs (capteurs de présence, contacteurs, capteurs de fumée, ...). DiaSuite apparaît donc comme un outil adapté et complet pour la simulation de ce système.

Pour la description des étapes suivantes, on ne rentre pas dans les détails d'installation ni de mise en place de l'environnement. Seul quelques explications sont données pour comprendre certains choix.

1. Définition du design

Les appareils

Pour le moment, le système de surveillance a à sa disposition :

- des contacteurs (pour les portes et fenêtres),
- des capteurs de fumée,
- une alarme,
- un écran (pour afficher des informations dans la zone à sécuriser),
- un iButton (pour identifier les personnes).

Chacun de ces appareils possède leurs propres caractéristiques et sont présents dans une ou plusieurs salle de la zone à sécuriser (dans notre cas, il n'y a qu'une seule salle mais s'il y a plusieurs salles, notre application est ainsi adaptée). Chaque appareil hérite alors d'un LocatedDevice qui présente l'emplacement en attribut.

Le design est décrit dans un fichier .diaspec présent dans le package *fr.inria.phoenix.scenario.airroom.spec* et présente toute l'architecture du système. Un langage spécifié est utilisé mais facilement compréhensible (voir manuel).

Voici quelques lignes sur certains appareils pour mieux comprendre ce langage.

Pour le design de LocatedDevice :

```
device LocatedDevice {
    attribute location as String;
}
```

Ici, l'appareil « LocatedDevice » présente l'emplacement en attribut.

Pour le design de l'alarme :

```
action Alert {
    ring();
    stop();
}

device Alarm extends LocatedDevice {
    action Alert;
}
```

Ici, l'appareil « Alarm » hérite de LocatedDevice pour préciser son emplacement et peut réaliser les actions décrites dans « Alert » définit juste au-dessus.

Pour le design du détecteur de fumée :

```
enumeration SmokeState {
    NOK, OK
}

device SmokeSensor extends LocatedDevice {
    source smoke as SmokeState;
}
```

Ici, le capteur de fumée « SmokeSensor » hérite de LocatedDevice pour la même raison et possède la source* « smoke » de type « SmokeState » (représente les différents états du capteur ; NOK : état anormal, présence de fumée ; OK : état normal, pas de fumée).

*Une source permet d'envoyer des informations à un contexte, voir ci-dessous.

Le contexte

Notre contexte « RoomContext », celui de la salle, rassemble toutes les sources des appareils qui envoient des informations sur leur état (contacteurs, capteurs et iButton). De plus, ce contexte permet

d'envoyer au contrôleur (voir ci-après) l'état courant de la salle, après avoir analyser les informations qu'il a reçu. Voici son design :

```
enumeration RoomState {
    ALERT_SMOKE, ALERT_ID, REST_CONTACT, REST_ID, REST_SMOKE, WAIT
}

context RoomContext as RoomState indexed by location as String {
    source smoke from SmokeSensor;
    source contact from DoorContactSensor;
    source contact from WindowContactSensor;
    source button from IButton;
}
```

L'opérateur « indexed by location » permet l'envoie de l'état de la salle dans la salle en question. Le type « RoomState » représente l'état de la salle :

- ALERT_SMOKE : salle en alerte à cause de la présence de fumée,
- ALERT_ID : salle en alerte à cause d'une identification invalide,
- REST_CONTACT : salle en repos par l'absence de personnes,
- REST_ID : salle en repos par l'identification acceptée d'un visiteur,
- REST_SMOKE : salle en repos par l'absence de fumée,
- WAIT : salle en attente d'une identification.

Le contrôleur

Notre contrôleur reçoit l'état de la salle par le contexte. Il établit ainsi les actions à effectuer sur les sorties en fonction de cet état. Ces actions sont les suivantes :

- faire sonner et arrêter l'alarme,
- rendre le iButton visible ou non (pour la simulation, il est plus cohérent que s'il n'y a personne dans la salle, l'identification n'est pas possible)
- afficher des informations sur l'écran de la TV.

Voici son design :

```
controller RoomController {
    context RoomContext;
    action Alert on Alarm;
    action EnableIButton on IButton;
    action Display on Screen;
}
```

Une fois le fichier .diaspec édité, il faut lancer la construction du framework. Puis passer à l'étape d'implémentation.

2. Implémentation des classes de appareils, contextes et contrôleurs

Voir manuel METTRE LIENS pour plus de détails.

Lors de cette étape, le développeur doit implémenter les classes de ces appareils, contextes et contrôleurs. Chacune de ces classes hérite de sa classe abstraite correspondante générée dans le framework. Ainsi, à chaque re-génération du framework, les fichiers développés par le développeur ne sont pas supprimés ni modifiés. Il n'a ensuite plus qu'à coder les méthodes abstraites de ces classes.

Le contexte

Dans notre cas, nous n'avons qu'un seul contexte « *RoomContext* », celui de la salle. Le contexte « *RoomContextImpl* » à implémenter hérite donc de « *RoomContext* ».

Premièrement, il faut initialiser toutes les sources avant de les utiliser :

```
public void postInitialize() {
    allDoorContactSensors().subscribeContact();
    allWindowContactSensors().subscribeContact();
    allSmokeSensors().subscribeSmoke();
    allIButtons().subscribeButton();
}
```

Chacune de ces sources possède une méthode lui étant propre. Cette méthode permet de gérer l'état de la salle en fonction de l'état de la source en question. Ces méthodes sont les suivantes :

```
public void onNewContact(DoorContactSensorProxy proxy,ContactState newContactValue) {}
public void onNewContact(WindowContactSensorProxy proxy, ContactState newContactValue) {}
public void onNewSmoke(SmokeSensorProxy proxy, SmokeState newSmokeValue) {}
public void onNewButton(IButtonProxy proxy, IButtonState newButtonValue) {}
```

On explique en détails la méthode « *onNewSmoke* ». L'évènement reçu par le contexte est constitué de :

- un proxy « *SmokeSensorProxy* » : instance de l'appareil « *SmokeState* »
- un « *SmokeState* » : valeur de l'évènement, donc ici, état du « *SmokeState* »

Tout d'abord, le contexte récupère l'emplacement du « *SmokeState* » pour pouvoir ensuite envoyer l'état à la salle où est situé l'appareil :

```
String location = proxy.getLocation();
```

Maintenant, il envoie l'état de salle en fonction de la valeur de l'évènement. L'envoi de l'état à la salle se fait par la méthode « *setRoomContext* » qui prend en paramètre l'état qu'on veut envoyer et l'emplacement où on veut envoyer. Si l'état du «*SmokeState*» est OK, on vérifie en plus que la salle n'est pas déjà en état de repos par identification, afin d'éviter de renvoyer une information redondante à la salle. On a alors les lignes suivantes :

```
if (newSmokeValue == SmokeState.NOK) {
    setRoomContext(RoomState.ALERT_SMOKE, location);
}
else {
    if (getRoomContext(location) != RoomState.REST_ID) {
                setRoomContext(RoomState.REST_SMOKE, location);
    }
}
```

L'automate définit ici METTRE LIENS peut être développer dans cette classe pour une meilleure gestion des états.

Le contrôleur

Dans notre cas, nous n'avons qu'un seul contrôleur « *RoomController* », celui contrôlant l'état de la salle. Le contrôleur « *RoomControllerImpl* » à implémenter hérite donc « *RoomController* ».

Ce contrôleur permet de gérer les actions à réaliser sur certains appareils (ceux définis dans le design du contrôleur) en fonction de la valeur de l'état de la salle qui receptionne. Exemple : s'il reçoit l'état ALERT_SMOKE, il déclenche l'alarme, rend visible le iButton (pour permettre une identification qui arrêterais l'alarme) et affiche un message à l'écran de la TV correspondant à cet état. Voici les lignes correspondantes :

```
switch (newRoomContextValue) {
    [...]
    case ALERT_SMOKE :
        allAlarms().ring();
        allIButtons().visible(true);
        allScreens().display("Smoke detected => Alarm is RINGING !!!");
        break;
    [...]
```

3. Édition de la simulation ou phase de test

Voir manuel pour plus de détails.

On édite la simulation par « launchEdition » sur fois les étapes précédentes réalisées.

1. Premièrement, on choisi les icônes pour représenter au mieux les appareils.

2. Deuxièmement, on entre l'image représentant l'environnement ainsi que celle de ces murs (plan en 2D réaliser avec GoogleSketchup). Ensuite, sur cet environnement, il faut définir les différentes salles. Lors de cette étape, il faut également définir le nom de la simulation et la date.

3. Troisièmement, il faut placer les appareils et les personnes dans l'environnement. Puis sauvegarder.

A l'issue de cette édition, les classes générées peuvent être modifier pour personnaliser la simulation. Ces classes sont celle de chaque type d'appareil simulé présent dans la salle ainsi que les classes *MyAgentModel*, *MyContextModel* et *MyWorldModel*. Dans notre cas, nous modifions les classes des appareils simulés et la classe *MyAgentModel*.

La modification des classes d'appareils simulés permet de personnaliser l'affichage de ces appareils lors de la simulation. Par exemple, l'alarme sera représentée par un voyant allumé ou éteint, tandis que le capteur de fumée sera représenté par un bouton à 2 états (voir les classes pour plus de détails).

Dans *MyAgentModel*, nous avons ajouter des listeners aux personnes. Ainsi par exemple, lorsqu'une personne entre dans la salle, un stimulus est envoyé, le notifiant à un appareil spécifique. Ce stimulus est reçu par cet appareil qui peut modifier son état en fonction de la valeur reçue (voir les méthodes receiveStimulus() dans les appareils simulés).

4. Lancement de la simulation

Pour être lancée, la simulation doit initialiser les composants. Ceci doit être fait dans le package « *fr.inria.phoenix.scenario.airroom.deploy* » et dans sa classe main.

```
public class Main {
    public static void main(String[] args) {
        new RoomContextImpl(new RmiServiceConfiguration()).initialize();
        new RoomControllerImpl(new RmiServiceConfiguration()).initialize();
    }
}
```

La simulation peut ensuite être lancée par « launchSimulation ».

Voici l'état de la simulation lorsqu'une personne rentre dans la salle par la porte :